**Spring Framework**

It is a light weight open source framework to develop J2EE application. It also provides loosely coupling through Inversion of Control(IOC). When IOC is applied objects are passively given their dependencies at run time instead of creating the dependent objects for themselves.

**Spring Modules:**

* **Spring Core: -** Fundamental functionality of spring framework like IOC with BeanFactory interface. BeanFactory is an implementation of factory pattern that applies IOC to separate your application configuration and dependencies specification from actual application core.
* **Spring Context: -** BeanFactory makes spring as container. ApplicationContext makes Spring as framework. This module supports Internationalization, Application life cycle events, Validation, Enterprise services like Email, EJB Integration, Remoting, Scheduling, Integration with Velocity, Free Marker etc.
* **Spring AOP: -** Support Aspect Oriented Programming to define the common functionalities for the application like logging, transaction management, security etc.
* **Spring DAO: -** Provides separate JDBC code for connection with spring.
* **Spring ORM: -** Supports Integration with ORM frameworks like Hibernate, Ibatis.
* **Spring WebFlow: -** Support Integration of spring with struts.
* **Spring MVC: -** Supports MVC in spring.

**IOC**

Objects are given their dependencies at runtime by some external entity that co-ordinate each object in the system using Dependency Injection.

**Types:**

* **Setters Injection: -** Dependencies are configured through Bean’s Setter method.
* **Constructor Injection: -** Dependencies are injected through Bean’s Constructor.
* **Interface Injection: -** Dependencies are injected through interfaces. Ex -Avalon.

**BeanFactory Interface:**

* Present in org.springframework.beans.factory interface.
* Used to create and dispense beans.
* Mostly used as xml BeanFactory class to load the bean based on the definition define in the xml file.

**Method:**

* Object getBean(): - This factory method will instantiate the bean and begin setting the bean property using dependency injection.

**ApplicationContext Interface:**

* Present in org.springframework.context.applicationContext interface.
* ClassPathXmlApplicationContext Class: - Loads the context definition from xml file located in class path (inside src)
* FileSystemApplicationContext: - Loads the context definition from an xml file located in file system

**Apache Maven**

It is a project build tool which provides dependencies for the project and download from the repositories.

**3 programs:**

* POJO class with getter and setters.
* XML file which contains dependencies.
* Client program.

**Beans Scope**

When defining a bean we have an option for declaring a scope for the bean.

**Types:**

* **Singleton: -** It is a default scope where a single instance per IOC container.
* **Prototype: -** Bean definition have any number of object instance
* **Request: -** This scopes a bean definition to a Http request.
* **Session: -** This scopes a bean definition to a Http session.

**Injecting Collections**

We can inject list, set, map and properties.

**Injecting Another Bean**

We can inject one bean into another bean using ref attribute.

**Wiring**

It is an association of bean property to its components at runtime.

**Autowiring**

Automatic configuration of beans to its properties without using <property> or <constructor-arg>

**Types:**

* **autowire = no: -** where we will do the wiring using <property> or <constructor-arg>.
* **autowire = byName: -** tries to match and wire its properties with a beans defined by the same name in the configuration file.
* **autowire = byType: -** tries to match and wire a property of its type matches with exactly one of the bean names in the configuration file.
* **autowire = constructor: -** tries to match and wire its constructor arguments with exactly one of the beans name in the configuration file.
* **autowire = autoDeduct: -** where it is used to wire the bean either using by constructor or by type

**@autowired**

It is used to do autowiring using annotation

**Types:**

* @autowired on setter methods
* @autowired on properties
* @autowired on constructors

**@qualifier**

It is used along with @autowired to remove the confusion by specifying which exact bean will be wired

**Spring MVC**

It helps in building flexible and loosely coupled web application based on MVC architecture. Whenever a client sends a request to spring MVC application,

* The Dispatcher Servlet first receives the request. Which acts as controller, it is http servlet configured in web.xml.
* The Dispatcher Servlet consults the handler mapping and invokes the controller associated with it.
* The controller is a class which does some business logic and returns ModelAndView object. The ModelAndView objects contain the Model data and View name.
* The Dispatcher Servlet sends the View name to a ViewResolver to find the actual view to invoke.
* The Dispatcher Servlet will pass the Model object to the View to render the result.

**Steps:**

* Create Dynamic web project and configure it to Maven project.
* Configure Dispatcher Servlet in web.xml.
* Configure the mapping in xml file as servlet\_name-servlet.xml in WEB-INF folder which contains all the mapping for the request. Default we use BeanNameUrlHandlerMapping where it maps the URL pattern with the name of the bean to invoke its related controller class to do business logic.
* Controller is a class which contains business logic and returns ModelAndView which contains Model data and View name.

**NOTES**

We use p:formView for injecting the formView property at http GET request, the formView will be rendered using p namespace the properties can be supplied using attributes rather than elements. Usually a form will be associated with a particular Model object, in the spring it is called as Command Object, to refer the Command Object in the jsp page we need to set a Command class with the help of setCommandClass() method.

**Hibernate**

It is a persistent framework used to persist the data into the database. It is an open source light weight ORM based framework to communicate with the database. ORM is a programming technique that maps the POJO to the columns of table in database.

**4 Programs:**

* **POJO Class or Persistent Class: -** it contains the properties and its getters and setters based upon the columns of the table.
* **Hibernate.cfg.xml file: -** it contains the information about the database properties like
* **hibernate.connection.driver\_class: -** represents JDBC driver name
* **hibernate.connection.url: -** contains JDBC Connection URL
* **hibernate.connection.username: -** represents database username
* **hibernate.connection.password: -** represents database password
* **hibernate.connection.pool-size: -** maximum number of pooled connection
* **hibernate.dialect: -** the class name of a hibernate org.hibernate.dialect.Dialect which allows hibernate to generate SQL query for a particular relational database.
* **hibernate.show\_sql: -** write all generated SQL statement in the console. Default is false.
* **hibernate.format\_sql: -** format the generated SQL on the console. Default is false.
* **hibernate.use\_sql\_comments: -** hibernate will generate comments inside SQL for easier debugging. Default is false.
* **hibernate.hbm2ddl.auto: -** automatically validates or export Schema DDL to the database. Values are: -
* **validate: -** validates the Schema makes no changes to the database.
* **update: -** if table not present it then creates else it will just insert.
* **create: -** creates the Schema destroying the previous data.
* **create-drop: -** drop and create a clean database each time we deploy.
* **hibernate.mapping: -** gives the information about the mapping file to the database.
* **Hibernate mapping file: -** used to provide the mapping between the persistent object to the column of the table. Tags are as follows: -
* **<class>: -** represents the persistent class, dynamic-insert=true/false which means exclude the null property value in SQL insert statement, dynamic-update = true excludes the unmodified property in the SQL update statement,
* **<id>: -** represents the primary key for the column
* **<generator>: -** used to provide the value for primary key or auto generated using attribute class = ”assigned/auto/identity/native/hilo/foreign/seqhilo/uuid/guid”
* **<property>: -** used to map the other columns; type = “the data type of property”; unique = “true”; not null = “true” ; size = “100”
* **Main Class: -** it is used to store the data into the database for that we use four interfaces in org.hibernate.\*package
* **Configuration class: -** allows the application to access the database properties and mapping documents to be used when creating SessionFactory.
* **SessionFactory interface: -** used to create a session instance
* **Session interface: -** the main interface between java application and hibernate and used to create read and delete operations for instance of mapped entity classes. It has three states: -
  + **Transient: -** never persistent not associated with any session
  + **Persistent: -** associated with UNIX session
  + **Detached: -** previously persistent, now not associated with any Session

**Methods: -**

* **save(): -** it stores an object into database and will fail if the primary key is already persistent.
* **persist(): -** same like save() but save() returns Serializable Objects that is primary key whereas persist() returns void. Sop(session.save(emp)) will return primary key but Sop(session.persist(emp)) will give an error.
* **saveOrUpdate(): -** calls save() or update() based on operation, if the identifier exists, it will call update(), else it will call save().
* **update(): -** used for updating the object using identifier, if the identifier is missing or doesn’t exist it will throw exception.
* **delete(): -** removes the persistent instance from data source based on the identifier.
* **get(): -** used to retrieve the objects from database, it always hits the database and returns the real object, returns null if no row is found. Example -> Employee e = (Employee) session.get(Employee.class,”100”).
* **load(): -** used to retrieve the object, if no row found throws ObjectNotFound Exception. Example -> Employee e = (Employee) session.load(Employee.class,”100”).
* **clear(): -** clears the session.
* **flush(): -** flushing the session simply makes the data that is currently in the session synchronized with the database.
* **refresh(): -** used to reload an object
* **Transaction interface: -** used to specify the atomic unit of work

**Methods: -**

* + **commit()**
  + **rollback()**

**Hibernate Annotations**

It is used instead of hbm mapping file.

* **Persistent class: -** which contains the annotation used to map the persistent object with columns of the table present in javax.persistence.\* package.

**Annotations: -**

* **@Entity: -** represents the class is a persistent class
* **@Table: -** optional annotation used to map the persistent class with the table. Example - @Table(name = ”emp100”)
* **@ID: -** Example – @Id(column = “emp\_id”)
* **@GeneratedValue: -** Example - @GeneratedValue(strategy = GenerationType.AUTO/IDENTITY/TABLE/SEQUENCE)
* **@Column: -** Example – @Column(name = “emp\_name”)
* **@Temporal: -** Example – @Temporal(TemporalType.DATE/TIME/TIMESTAMP)
* **@Enumerated: -** Example – @Enumerated(EnumerationType.ORDINAL/STRING)

**Hibernate Inheritance**

In hibernate the persistence class can also be inherited.

**Types:**

* **Table per Class Inheritance: -** where a single table is created for all persistence class.
* **Table per Concrete Class Inheritance:** - where the table will be created for individual persistence class.
* **Table per Subclass Inheritance: -** where the table are linked using primary key and foreign key relationship

**Hibernate Query Language**

It is used to query the persistent classes instead of database using org.hibernate.Query interface.

**Types:**

* from Employee: - select everything from employee class.
* from Employee as e: - aliases
* select e.firstName, e.lastName from Employee e: - selects specified properties from Employee class
* from Employee e where e.id > 10: - all properties having id greater than 10
* from Employee e where e.id > 10 order by e.sal asc/desc: - orders the output
* using Named Parameter: - from Employee e where e.id = :a
* void setParameter(“a”,100): - set the named parameter :a as 100
* Query setFirstResult(int start): - takes an integer that represents the first row in your ResultSet, starts from 0
* Query setMaxResult(int start): - retrieves a fixed number of objects
* Select count(e.firstName) from employee e
* Query q = s.createQuery(“from Employee”)
* List l = q.list()
* Query q = s.createQuery(“from Employee e where e.id = 10”)
* Employee e1 = (Employee) q.UniqueResult()

**Hibernate Criteria Query Language**

It is used to build up a criteria query object programmatically where we can apply filtration and logical condition using org.hibernate.Criteria Interface.

**Types:**

* Criteria c = s.createCriteria(Employee.class)
* List l = c.list()
* Criteria c1 = s.createCriteria(Employee.class)
* c1.addRestriction.eq(“salary”,10000)
* List l = c1.list()
* c1.addRestrictions.gt(“salary”,100000)
* c1.addRestrictions.lt(“salary”,100000)
* c1.addRestrictions.like(“name”,”IOAN GUFFUDD”)
* c1.addOfRestrictions.iLike(“name”,”IOAN GUFFUDD”)
* c1.addRestrictions.between(“salary”,1000,2000)
* c1.addOfRestrictions.ge(“salary”,1000)
* c1.addOfRestrictions.le(“salary”,1000)
* c1.addOfRestrictions.ne(“salary”,1000)
* c1.addOfRestrictions.isNull(“salary”)
* c1.addOfRestriction.isNotNull(“salary”)
* c1.addOfRestriction.isEmpty(“salary”)
* c1.addOfRestriction.isNotEmpty(“salary”)
* Criteria c2 = s.createCriteria(Employee.class)
* Criteria c3 = Restriction.gt(“salary”,1000)
* Criteria c3 = Restriction.lt(“salary”,1000)
* logicalExpression l = Restriction.or(c3,c4)
* c2.add(l)
* logicalExpression l1 = Restriction.And(c3,c4)
* c2.add(l1)
* Criteria c5 = s.createCriteria(Employee.class)
* C5.add(Restriction.gt(“salary”,2000))
* C5.addOrder(order.desc(salary))
* C5.addOrder(order.asc(salary))

**Projections**

The Criteria API provide org.hibernate.criterion.Projections class:used to apply avg(), sum(), min(), max() in Criteria instead of HQL

* Criteria c6 = s.createCriteria(Employee.class)
* c6.setProjection(Projections.AVG(“salary”))
* c6.setProjection(Projections.SUM(“salary”))
* c6.setProjection(Projections.MIN(“salary”))
* c6.setProjection(Projections.MAX(“salary”))
* c6.setProjection(Projections.COUNT(“salary”))
* c6.setProjection(Projections.COUNT\_DISTINCT(“salary”))

**Hibernate Mappings**

* **One to One Mapping: -**

**Lazy fetching: -** FetchType.EAGER and FetchType.LAZY is used for collection if we set FetchType.LAZY then until we fetch the collection, the collection will not be loaded if we set FetchType.EAGER the collection will be loaded at the time of entity loading

**Spring ORM Hibernate**

* Create a dynamic project
* Copy the jar files in lib folder
* Configure the dispatcher servlet in web.xml
* Configure the mapping information in xml file
* Configure the mapping information in xml file(We need to configure <init-param> when we change the name and/or location of the xml file).
* <mvc:annotation-driven> tag is used to create Spring MVC with annotation

**Spring Security Module**

Spring security is a framework that provides security solution, handling authentication and authorization at both the web request level and the method level. It is used for: -

* Only authorized user should be able to access the application
* Unauthorized user should be presented with login screen
* Successful credentials should forward to the system and unsuccessful credential should forward to the login application

The module included are: -

* Spring-security-core
* Spring-security-taglib
* Spring-security-web

In spring security DelegatingFilterProxy provides a link between web.xml and application context. The filter name SpringSecurityFilterChain configures the basic structure of security in the xml file.

<http>: - includes configuration related url level security

<auto-config = “true”>: - includes some basic services like form-login, http-basic, logout

<use-expression>: - to use expression to secure individuals URLs

<intercept-url>: - this will match the requested url pattern from request and decide what action to take based on access value

<form-login>: - this will come into picture when user will try to access any secured URL. A login page mapped to login-page attribute will be served for authentication check. If not provided Spring will provide an inbuilt login page to an URL. It also contains an attribute for default-target-url if login success and authentication-failure-url if login fails

<logout>: - help to find the next view if logout is called.

<authentication-manager>: - is used to validate the username and password.

By default spring auto generates and configure a UserNamePasswordAuthenticationFilterBean.

This filter by default respond to the URL j\_spring\_security\_check when a processing a login post request for username field j\_username and password field j\_password.

UserDetailService interface is used in order to look up username and password and grant authorities for any given user. The interface provided with one method

UserDetails LoadUserByUserName (String username)throws UsernameNotFoundException

Here user details is the container for user core information

**Installation:**

Provided with one jar file log4j-1.2.144.jar

**Components:**

* **Logger**: Logger is a class in org.apache.log4j.\* package. We have to initialize one logger object for each java class like static Logger l = Logger.getLogger(className.class). the Logger class has some methods info(), debug(), warn(), fatal(), error()
* **Label**: Label is a class in org.apache.log4j.\* package. Each Label has priority order like debug <info <warn <error <fatal. fatal is the highest priority error.
* **Appender**: Used to write messages into a file or database or console. The different types of Appenders are: FileAppender, ConsoleAppender, JdbcAppender, SocketAppender etc.
* Layout: Used to define the formatting in which logs will print in a Repository. The different Layouts are SimpleLayout, PatternLayout, XmlLayout, HtmlLayout.

The output contains the time elapsed from the start of the program milliseconds, the thread name, the type of logger label, class name and the log messages. The root logger is the one that resides on the top of the logger hierarchy. We set its label to WARN and added the ConsoleAppender CAA. We use a layout called pattern layout for ConsoleAppender CAA. The pattern layout uses ConversionPattern to format the message like %r prints elapsed time in milliseconds, [%t] thread name, %p is the Priority of the logging event, %c prints the class name %x is used the output the nested diagnostic content associated with the thread that generate d the login event, %m use to output the application supplied message associated with the login event, %n outputs the platform dependent line separator character, %d used to output the date of login event like %d{HH:mm:ss, SSS}.